1. Tempo disponibile 120 minuti.
2. Non è possibile consultare appunti, slide, libri, persone, siti web, ecc.
3. Scrivere in modo leggibile, su ogni foglio, nome, cognome e numero di matricola.
4. Le soluzioni agli esercizi che richiedono di progettare un algoritmo devono:
   - spiegare a parole l'algoritmo (se utile, anche con l'aiuto di esempi o disegni),
   - fornire e commentare lo pseudo-codice (indicando il significato delle variabili),
   - calcolare la complessità (con tutti i passaggi matematici necessari),
   - se l'esercizio ammette più soluzioni, a soluzioni computazionalmente più efficienti e/o concettualmente più semplici sono assegnati punteggi maggiori.

1. Calcolare la complessità $T(n)$ del seguente algoritmo `mystery1`:

```
algoritmo mystery1(n: Int) --> Int
    if (n <= 1)
        return 32
    else
        return mystery2(n/2) + mystery1(n/2)
    endif

algoritmo mystery2(m: Int) --> Int
    if (m == 1)
        return 2
    else
        return 2 * mystery2(m-1)
    endif
```

**Soluzione** L'algoritmo `mystery1` utilizza `mystery2`. Iniziamo quindi l'analisi da tale secondo algoritmo, che risulta essere un algoritmo ricorsivo con complessità $T'(m)$ caratterizzata dalla seguente equazione di ricorrenza:

$$T'(m) = \begin{cases} c_1 & \text{se } m \leq 1 \\ T'(m-1) + c_2 & \text{altrimenti} \end{cases}$$

con $c_1$ e $c_2$ costanti. Si noti che tale equazione non si basa su partizionamenti bilanciati e quindi non è possibile l'utilizzo del Master Theorem. Procediamo con il metodo dell'iterazione:

$$T'(m) = T'(m-1) + c_2 = T'(m-2) + c_2 + c_2 = \cdots = T'(1) + (m-1) \times c_2 = c_1 + (m-1) \times c_2 = O(m)$$

Analizziamo ora `mystery1`. Tutte le operazioni hanno costo costante ad esclusione della invocazione a `mystery2` e dell'invocazione ricorsiva a `mystery1`. Il tempo di calcolo $T(n)$ soddisfa quindi la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ T(n/2) + O(n/2) & \text{altrimenti} \end{cases}$$

Applichiamo il Master Theorem. Abbiamo $\alpha = \frac{\log 1}{\log 2} = 0$ e $\beta = 1$. Visto che $\alpha < \beta$, per il terzo caso del teorema abbiamo $T(n) = O(n^\beta) = O(n)$.

2. Scrivere un algoritmo che prende in input un albero binario $T$ e un intero positivo $k$ e calcola il numero di nodi che si trovano esattamente a profondità $k$ (nota: la radice si trova a profondità zero). Discutere la complessità nel caso pessimo e ottimo.

**Soluzione** Per risolvere l'esercizio si può usare una visita in profondità partendo dalla radice e passando come parametro l'altezza $k$. Ad ogni chiamata ricorsiva si decrementa il valore di $k$ fino al valore 0; in questo caso il nodo raggiunto sarà a livello $k$ per cui dovrà essere conteggiato e la visita può essere interrotta.

```
nodiK(node T, int k ) --> int
   if ( T == null )
      return 0;
   else
      if (k == 0)
         return 1;
      else
         return nodiK(T.left, k-1) + nodiK(T.right, k-1);
      endif
   endif
```

Il caso ottimo si verifica se l'albero degenera in una lista (quindi se ci sono solo figli a sinistra o solo a destra). In questo caso la complessità è $O(k)$ visto che è sufficiente attraversare $k$ nodi. Il caso pessimo si verifica se l'albero è completo fino al livello $k$ per cui è necessario attraversare tutti i nodi fino a quel livello. Visto che un albero binario completo di livello $k$ ha $2^{k+1} - 1$ nodi, la complessità risulta essere $O(2^k)$.

Una soluzione alternativa avrebbe potuto sfruttare una visita in ampiezza: ogni volta che si aggiunge un nodo figlio alla coda usata per la visita si memorizza la coppia $[nodo, livello]$, dove il livello si calcola dal livello del nodo corrente aumentato di 1. Dopo aver estratto (e contato) tutti i nodi a livello $k$ la visita può essere interrotta.

3. Una cisterna contiene rifiuti gassosi tossici da smaltire. Lo smaltimento si effettua riempiendo particolari contenitori, ognuno avente una propria capacità possibilmente differente dalle capacità degli altri contenitori. Quando si collega la cisterna ad un contenitore per lo smaltimento del gas, il trasferimento del gas prevede di riempire completamente il contenitore, a meno che la cisterna non si svuoti completamente prima di terminare il riempimento del contenitore. Bisogna distribuire il gas nel numero massimo di contenitori possibili. Progettare un algoritmo che riceve in input la quantità $K$ di metri cubi di gas inizialmente nella cisterna, ed un vettore $V$ di lunghezza $n$ tale che $V[i]$ è la capacità in metri cubi del $i$-esimo contenitore (i contenitori disponibili sono $n$). Scrivere un algoritmo che restituisce un vettore $O$ di lunghezza $n$ tale the $O[i]$ è la quantità di gas che verrà smaltita nel $i$-esimo contenitore. Si ricordi che lo smaltimento deve utilizzare più contenitori possibili. Potete assumere che la sommatoria delle capacità dei contenitori sia superiore alla quantità di gas da smaltire.

**Soluzione** Si può procedere secondo un criterio greedy, riempiendo i contenitori di capacità inferiore senza utilizzare (a meno che non sia necessario) quelli di capacità superiore. Questo permette di massimizzare il numero di contenitori utilizzati. Per sapere quali contenitori usare, si può utilizzare una coda con priorità in cui si inseriscono tutte le capacità disponibili, con associato il loro indice di posizionamento nel vettore in input $V$. Una volta riempita la coda con priorità, si estraggono in ordine crescente di capacità i vari contenitori, riempiendoli se c'è gas residuo nella cisterna.

L'algoritmo `RiempiContenitori` utilizza la coda con priorità $Q$ e la variabile `res` indicante la quantità residua di gas da smaltire.

```
algoritmo RiempiContenitori(K: number, V:number[1..n]) --> number[1..n]

    // dichiarazione variabili
    number res = K; O[1..n]
    int j
    CodaPriorita<int, number> Q

    // riempimento della coda con priorita'
    for i = 1..n
        Q.insert(V[i],i)
    endfor
```

```
// riempimento dei contenitori secondo ordine di estrazione dalla coda
for i = 1..n
    j = Q.find(); Q.deleteMin
    if (K>V[j]) then
        // il j-esimo contenitore si puo' riempire completamente
        O[j] = V[j]
        K -= V[j]
    elseif (K>0) then
        // il j-esimo contenitore non si puo' riempire completamente
        O[j] = K
        K = 0
    else
        // il gas e' gia' stato tutto smaltito quindi K=0
        O[j] = K
    endif
endfor

// restituisce il vettore in output O
return O
```

L'algoritmo esegue operazioni di costo costante ad esclusione delle $n$ operazioni di inserimento e delle $n$ operazioni di cancellazione dalla coda con priorità, presenti all'interno dei due cicli for. Assumendo una implementazione tramite heap, tali operazioni hanno un costo logaritmico nella dimensione dell'heap. La dimensione massima dell'heap coincide con il numero di contenitori $n$. Concludendo avremo un tempo di calcolo $T(n) = 2n \times O(\log n) = O(n \log n)$.

4. Progettare un algoritmo che, dato un grafo orientato pesato $G = (V, E, w)$, e due sottoinsiemi disgiunti $V_1, V_2 \subseteq V$ (disgiunti vuol dire che $V_1 \cap V_2 = \emptyset$), restituisce il cammino minimo da $V_1$ a $V_2$ ovvero restituisce il minimo cammino minimo da un qualsiasi vertice in $V_1$ ad un qualsiasi altro vertice in $V_2$.

**Soluzione** Considerando che dobbiamo confrontare tanti cammini minimi, risulta conveniente l'utilizzo dell'algoritmo di Floyd-Warshall per il calcolo di tutti i cammini minimi tra tutte le coppie di vertici di un grafo orientato pesato. Una volta calcolati tutti i cammini minimi, si controllano tutti quelli da un nodo $v_1 \in V_1$ ad un nodo $v_2 \in V_2$ per scegliere il più piccolo fra tutti questi. Una volta trovata la coppia $(v_1, v_2)$ con il cammino minore, si procede a stampare il relativo cammino.

```
algoritmo MinimoCamminoMinimo( G=(V,E,w), V1 subset V, V2 subset V )

    // esecuzione dell'algoritmo di Floyd-Warshall
    int n = G.numNodi()
    double D[1..n, 1..n]
    int x, y, k, next[1..n, 1..n]
    for x = 1..n do
        for y = 1..n do
            if (x == y) then
                D[x,y] = 0
                next[x,y] = -1
            elseif ((x,y) in E) then
                D[x,y] = w(x,y)
                next[x,y] = y
            else
                D[x,y] = INFINITY
                next[x,y] = -1
            endif
        endfor
    endfor
```

Cont.

```
    for k = 1..n do
        for x = 1..n do
            for y = 1..n do
                if (D[x,k] + D[k,y] < D[x,y]) then
                    D[x,y] = D[x,k] + D[k,y]
                    next[x,y] = next[x,k]
                endif
            endfor
        endfor
    endfor

    // ricerca del miglior cammino da un nodo v1 in V1 a un nodo v2 in V2
    int v1min, v2min, min = INFINITY
    for each v1 in V1
        for each v2 in V2
            if (D[v1,v2]<min) then
                min = D[v1,v2]
                v1min = v1
                v2min = v2
            endif
        endfor
    endfor

    // stampa del cammino minimo da v1min a v2min
    PrintPath(v1min, v2min, next)

// algoritmo standard per la stampa di un cammino secondo l'algoritmo di Floyd-Warshall
algoritmo PrintPath(int u, int v, int next[1..n, 1..n])
    if ( u != v and next[u,v] < 0 ) then
        errore(u e v non sono connessi)
    else
        print u
        while ( u != v ) do
            u = next[u,v]
            print u
        endwhile
    endif
```

La complessità dell'algoritmo è la medesima dell'algoritmo di Floyd-Warshall, ovvero $T(n, m) = O(n^3)$, dove $n$ è il numero di vertici ed $m$ il numero di archi nel grafo. Si noti infatti che le parti aggiuntive dell'algoritmo servono per ricercare il vertice `v1min` di inizio e di fine `v2min` del cammino minimo (con costo $O(n^2)$) e per stampare il cammino da `v1min` a `v2min` (con costo $O(n)$). Tali costi aggiuntivi sono inferiori in ordine di grandezza e vengono quindi assorbiti dal costo dell'algoritmo di Floyd-Warshall.

The End.